# Zookeeper

## 一、ZooKeeper的背景

### 1.1 认识ZooKeeper

ZooKeeper---译名为“动物园管理员”。动物园里当然有好多的动物，游客可以根据动物园提供的向导图到不同的场馆观赏各种类型的动物，而不是像走在原始丛林里，心惊胆颤的被动 物所观赏。为了让各种不同的动物呆在它们应该呆的地方，而不是相互串门，或是相互厮杀，就需要动物园管理员按照动物的各种习性加以分类和管理，这样我们才能更加放心安全的观赏动物。

回到企业级应用系统中，随着信息化水平的不断提高，企业级系统变得越来越庞大臃肿，性能急剧下降，客户抱怨频频。拆分系统是目前我们可选择的解决系统可伸缩性和性能问题的唯一行之有效的方法。但是拆分系统同时也带来了系统的复杂性——各子系统不是孤立存在的，它们彼此之间需要协作和交互，这就是我们常说的分布式系统。各个子系统就好比动物园里的动物，为了使各个子系统能正常为用户提供统一的服务，必须需要一种机制来进行协调——这就是ZooKeeper（动物园管理员）。

### 1.2 为什么使用ZooKeeper

我们知道要写一个分布式应用是非常困难的，主要原因就是局部故障。一个消息通过网络在两个节点之间传递时，网络如果发生故障，发送方并不知道接收方是否接收到了这个消息。他可能在网络故障前就收到了此消息，也可能没有收到，又或者可能接收方的进程死了。发送方了解情况的唯一方法就是再次连接发送方，并向他进行询问。这就是局部故障：根本不知道操作是否失败。因此，大部分分布式应用需要一个主控、协调控制器来管理物理分布的子进程。目前，大部分应用需要开发私有的协调程序，缺乏一个通用的机制。协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器。协调服务非常容易出错，并很难从故障中恢复。例如：协调服务很容易处于竞态1甚至死锁2。Zookeeper的设计目的，是为了减轻分布式应用程序所承担的协调任务。

Zookeeper并不能阻止局部故障的发生，因为它们的本质是分布式系统。他当然也不会隐藏局部故障。ZooKeeper的目的就是提供一些工具集，用来建立安全处理局部故障的分布式应用。

ZooKeeper是一个分布式小文件系统，并且被设计为高可用性。通过选举**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)**和集群复制可以避免单点故障3，由于是文件系统，所以即使所有的ZooKeeper节点全部挂掉，数据也不会丢失，重启服务器之后，数据即可恢复。另外ZooKeeper的节点更新是原子的，也就是说更新不是成功就是失败。通过版本号，ZooKeeper实现了更新的乐观锁4，当版本号不相符时，则表示待更新的节点已经被其他客户端提前更新了，而当前的整个更新操作将全部失败。当然所有的一切ZooKeeper已经为开发者提供了保障，我们需要做的只是调用API。与此同时，随着分布式应用的的不断深入，需要对集群管理逐步透明化监控集群和作业状态，可以充分利ZK的独有特性。

### 1.3 ZooKeeper的应用

ZooKeeper本质上是一个分布式的小文件存储系统。原本是Apache [Hadoop](http://lib.csdn.net/base/hadoop)的一个组件，现在被拆分为一个Hadoop的独立子项目，在[Hbase](http://lib.csdn.net/base/hbase" \o "Hbase知识库" \t "_blank)（Hadoop的另外一个被拆分出来的子项目，用于分布式环境下的超[大数据](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "_blank)量的DBMS）中也用到了ZooKeeper集群。

Hadoop，使用Zookeeper的事件处理确保整个集群只有一个NameNode，存储配置信息等.  
　　HBase，使用Zookeeper的事件处理确保整个集群只有一个HMaster，察觉HRegionServer联机和宕(dàng)机，存储访问控制列表等。

有人会怀疑ZooKeeper的执行能力，在ZooKeeper诞生的地方——Yahoo!他被用作雅虎消息代理的协调和故障恢复服务。雅虎消息代理是一个高度可扩展的发布-订阅系统，他管理着成千上万台联及程序和信息控制系统。它的吞吐量标准已经达到大约每秒10000基于写操作的工作量。对于读操作的工作量来说，它的吞吐量标准还要高几倍。

## 二、ZooKeeper的介绍

### 2.1 ZooKeeper的概述

Zookeeper 是为分布式应用程序提供高性能协调服务的工具集合，也是Google的Chubby一个开源的实现，是Hadoop 的分布式协调服务。它包含一个简单的原语集5，分布式应用程序可以基于它实现配置维护、命名服务、分布式同步、组服务等。Zookeeper可以用来保证数据在ZK集群之间的数据的事务性一致6。其中ZooKeeper提供通用的分布式锁服务7，用以协调分布式应用。

Zookeeper作为Hadoop项目中的一个子项目,是 Hadoop集群管理的一个必不可少的模块,它主要用来解决分布式应用中经常遇到的数据管理问题，如集群管理、统一命名服务、分布式配置管理、分布式消息队列、分布式锁、分布式协调等。在Hadoop中，它管理Hadoop集群中的NameNode，还有在Hbase中Master Election、Server 之间状态同状步等。  
　　Zoopkeeper 提供了一套很好的分布式集群管理的机制，就是它这种基于层次型的目录树的**[数据结构](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)**，并对树中的节点进行有效管理，从而可以设计出多种多样的分布式的数据管理模型。

### 2.2 ZooKeeper的设计目标

众所周知，分布式环境下的程序和活动为了达到协调一致目的，通常具有某些共同的特点，例如，简单性、有序性等。ZooKeeper不但在这些目标的实现上有自身特点，并且具有独特优势。下面我们将简述ZooKeeper的设计目标。

#### （1）简单化

ZooKeeper允许各分布式进程通过一个共享的命名空间相互联系，该命名空间类似于一个标准的层次型的文件系统：由若干注册了的数据节点构成(用Zookeeper的术语叫znode)，这些节点类似于文件和目录。典型的文件系统是基于存储设备的，文传统的文件系统主要用于存储功能，然而ZooKepper的数据是保存在内存中的。也就是说，可以获得高吞吐和低延迟。ZooKeeper的实现非常重视高性能、高可靠，以及严格的有序访问。

高性能保证了ZooKeeper可以用于大型的分布式系统，高可靠保证了ZooKeeper不会发生单点故障，严格的顺序访问保证了客户端可以获得复杂的同步操作原语。

#### （2）健壮性

就像ZooKeeper需要协调的分布式系统一样，它本身就是具有冗余结构，它构建在一系列主机之上，叫做一个”ensemble”。  
　　构成ZooKeeper服务的各服务器之间必须相互知道，它们维护着一个状态信息的内存映像8，以及在持久化存储中维护着事务日志和快照9。只要大部分服务器正常工作，ZooKeeper服务就能正常工作。  
　　客户端连接到一台ZooKeeper服务器。客户端维护这个TCP连接，通过这个连接，客户端可以发送请求、得到应答，得到监视事件以及发送心跳。如果这个连接断了，客户端可以连接到另一个ZooKeeper服务器。

#### （3）有序性

ZooKeeper给每次更新附加一个数字标签，表明ZooKeeper中的事务顺序，后续操作可以利用这个顺序来完成更高层次的抽象功能，例如同步原语7。

#### （4）速度优势

ZooKeeper特别适合于以读为主要负荷的场合。ZooKeeper可以运行在数千台机器上，如果大部分操作为读，例如读写比例为10:1，ZooKeeper的效率会很高。

### 2.3 ZooKeeper的集群

ZK集群如下图2.1所示。这是实际应用的一个场景，该ZooKeeper集群当中一共有5台服务器，有两种角色Leader和Follwer，5台服务器连通在一起，客户端有分别连在不同的ZK服务器上。如果当数据通过客户端1，在左边第一台Follower服务器上做了一次数据变更，他会把这个数据的变化同步到其他所有的服务器，同步结束之后，那么其他的客户端都会获得这个数据的变化。

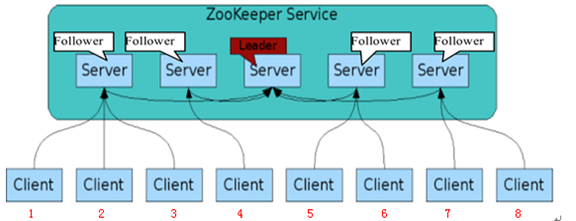


图 2.1

**注意：**

通常Zookeeper由2n+1台servers组成，每个server都知道彼此的存在。每个server都维护的内存状态镜像以及持久化存储的事务日志和快照。为了保证Leader选举能过得到多数的支持，所以ZooKeeper集群的数量一般为奇数。对于2n+1台server，只要有n+1台（大多数）server可用，整个系统保持可用。

#### 2.3.1 集群中的角色

在ZooKeeper集群当中，集群中的服务器角色有两种Leader和Learner，Learner角色又分为Observer和Follower，具体功能如下：

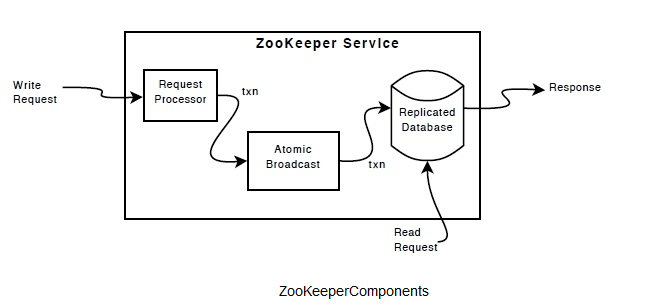
1.领导者(leader)，负责进行投票的发起和决议，更新系统状态

2.学习者(learner)，包括跟随者（follower）和观察者（observer），

3.follower用于接受客户端请求并向客户端返回结果，在选主过程中参与投票

4.Observer可以接受客户端请求，将写请求转发给leader，但observer不参加投票过程，只同步leader的状态，observer的目的是为了扩展系统，提高读取速度。

5. 客户端(client)，请求发起方



ZooKeeper的组件图中给出了ZooKeeper服务的高层次的组件。除了请求处理器（requestprocessor）外，构成ZooKeeper服务的每个服务器都有一个备份。复制的**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)**（replicateddatabase）是一个内存数据库，包含整个数据树。为了可恢复，更新会被log到磁盘，并且在更新这个内存数据库之前，先序列化到磁盘。

每个ZooKeeper都为客户端提供服务。客户端只连接到一个服务器，并提交请求。读请求直接由本地的复制数据库提供数据。对服务状态进行修改的请求、写请求通过一个约定的协议进行通讯。

作为这个协议的一部分，所有的写请求都被传送到一个叫“首领(leader)”的服务器，而其他的服务器，叫做“(随从)followers”，follower从leader接收信息修改的提议，并同意进行。当leader发生故障时，协议的信息层（messaginglayer）关注leader的替换，并同步到所有的follower。

ZooKeeper采用一个自定义的信息原子操作协议，由于信息层的操作是原子性的，ZooKeeper能保证本地的复制数据库不会产生不一致。当leader接收到一个写请求，它计算出写之后系统的状态，把它变成一个事务。

#### 2.3.2 Zookeeper的读写机制和保证及特点

##### （1）ZooKeeper的读写机制

Zookeeper是一个由多个server组成的集群

一个leader，多个follower

每个server保存一份数据副本

全局数据一致

分布式读写

更新请求转发，由leader实施

##### （2）ZooKeeper的保证

ZooKeeper运行非常快而且简单。虽然它的目标是构建更加复杂服务（例如同步）的基础，但它提供了一些保证，如下：

1.顺序一致性：来自于客户端的更新，根据发送的先后被顺序实施。

2.唯一的系统映像：尽管客户端连接到不同的服务器，但它们看到的一个唯一（一致性）的系统服务，client无论连接到哪个server，数据视图都是一致的。

3.可靠性：一旦实施了一个更新，就会一直保持那种状态，直到客户端再次更新它，同时数据更新原子性，一次数据更新要么成功，要么失败。

4.及时性：在一个确定的时间内，客户端看到的系统状态是最新的。

##### （3）ZooKeeper特点

最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

可靠性：具有简单、健壮、良好的性能，如果消息m被一台服务器接受，那么它将被所有的服务器接受。

实时性：Zookeeper保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。 但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

等待无关(wait-free)：慢的或者失效的client，不得干预快速的client的请求，使得每个client都能有效的等待。

原子性：更新只能成功或者失败，没有中间状态。

顺序性：包括全局有序和偏序两种：

全局有序：是指如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布；

偏序：是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面

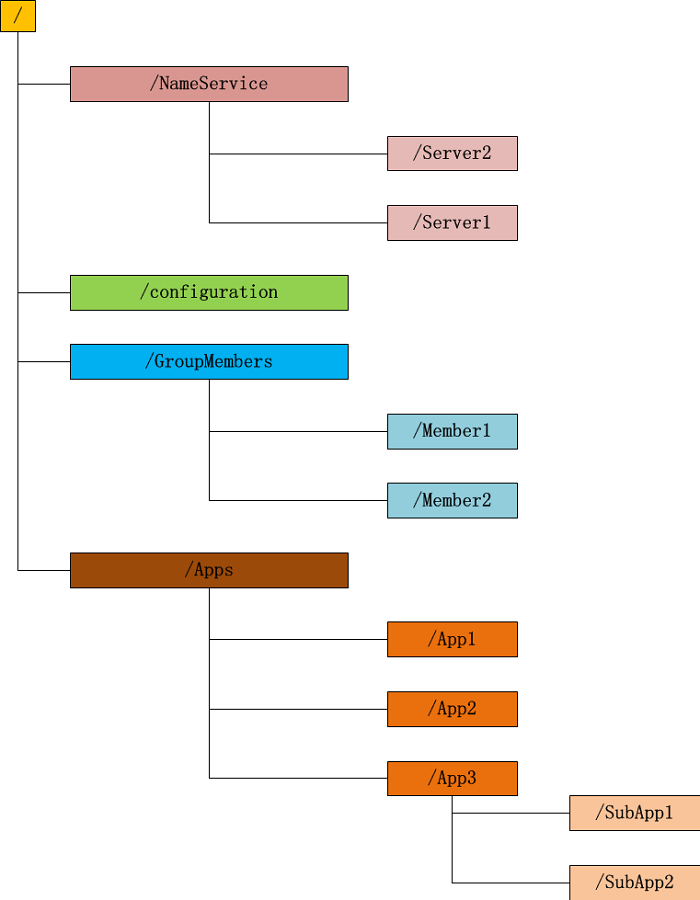
## 三、ZooKeeper服务

### 3.1 ZooKeeper数据模型

ZooKeeper拥有一个层次的命名空间，这个和分布式的文件系统非常相似。不同的是ZooKeeper命名空间中的Znode，兼具文件和目录两种特点。既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分，并可以具有子znode。用户对znode具有增、删、改、查等操作（权限允许的情况下）。

znode具有原子性操作，每个znode的数据将被原子性地读写，读操作会读取与znode相关的所有数据，写操作会一次性替换所有数据。zookeeper并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以KB为大小单位。zooKeeper的服务器和客户端都被设计为严格检查并限制每个znode的数据大小至多1M，当时常规使用中应该远小于此值。

Zonde由路径标注，ZooKeeper中被表示成有反斜杠分割的Unicode字符串，如同Unix中的文件路径。路径必须是绝对的，因此他们必须由反斜杠来字符开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。ZooKeeper的数据结构, 与普通的文件系统极为类似. 见下图：



图中的每个节点称为一个znode. 每个znode由3部分组成:

1.stat：此为状态信息, 描述该znode的版本, 权限等信息.

2.data：与该znode关联的数据.

3.children：该znode下的子节点.

#### 3.1.1 ZooKeeper节点Znode

ZooKeeper目录树中每一个节点对应一个Znode。每个Znode维护着一个属性结构，它包含着版本号(dataVersion)，时间戳(ctime,mtime)等状态信息。ZooKeeper正是使用节点的这些特性来实现它的某些特定功能。每当Znode的数据改变时，他相应的版本号将会增加。每当客户端检索数据时，它将同时检索数据的版本号。并且如果一个客户端执行了某个节点的更新或删除操作，他也必须提供要被操作的数据版本号。如果所提供的数据版本号与实际不匹配，那么这个操作将会失败。

**Znode是客户端访问ZooKeeper的主要实体，它包含以下几个特征：**

##### （1）Watches

客户端可以在节点上设置watch(我们称之为监视器)。当节点状态发生改变时(数据的增、删、改)将会触发watch所对应的操作。当watch被触发时，ZooKeeper将会向客户端发送且仅发送一条通知，因为watch只能被触发一次。

##### （2）数据访问

ZooKeeper中的每个节点存储的数据要被原子性的操作。也就是说读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。

##### （3）节点类型

ZooKeeper中的节点有两种，分别为临时节点和永久节点。节点的类型在创建时即被确定，并且不能改变。  
　　ZooKeeper的临时节点：该节点的生命周期依赖于创建它们的会话。一旦会话结束，临时节点将被自动删除，当然可以也可以手动删除。另外，需要注意是，ZooKeeper的临时节点不允许拥有子节点。  
　　ZooKeeper的永久节点：该节点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才能被删除。

##### （4）顺序节点（唯一性的保证）

当创建Znode的时候，用户可以请求在ZooKeeper的路径结尾添加一个递增的计数。这个计数对于此节点的父节点来说是唯一的，它的格式为“%10d”(10位数字，没有数值的数位用0补充，例如“0000000001”)。当计数值大于232-1时，计数器将溢出。

org.apache.zookeeper.CreateMode中定义了四种节点类型，分别对应：

PERSISTENT：永久节点

EPHEMERAL：临时节点

PERSISTENT\_SEQUENTIAL：永久节点、序列化

EPHEMERAL\_SEQUENTIAL：临时节点、序列化

#### 3.1.2 ZooKeeper中的时间

ZooKeeper有多种记录时间的形式，其中包含以下几个主要属性：

##### （1）Zxid

致使ZooKeeper节点状态改变的每一个操作都将使节点接收到一个zxid格式的时间戳，并且这个时间戳全局有序。也就是说，也就是说，每个对节点的改变都将产生一个唯一的zxid。如果zxid1的值小于zxid2的值，那么zxid1所对应的事件发生在zxid2所对应的事件之前。实际上，ZooKeeper的每个节点维护者三个zxid值，为别为：cZxid、mZxid、pZxid。

cZxid： 是节点的创建时间所对应的Zxid格式时间戳。

mZxid：是节点的修改时间所对应的Zxid格式时间戳。

实现中zxid是一个64为的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个 新的epoch。低32位是个递增计数。

##### （2）版本号

对节点的每一个操作都将致使这个节点的版本号增加。每个节点维护着三个版本号，他们分别为：

version 节点数据版本号

cversion 子节点版本号

aversion 节点所拥有的ACL版本号

#### 3.1.3 节点的属性结构

通过前面的介绍，我们可以了解到，一个节点自身拥有表示其状态的许多重要属性，如下图所示。



#### 3.1.4 Zonde总结

（1）znode中的数据可以有多个版本，在查询该znode数据时就需要带上版本信息。如：set path version / delete path version

（2）znode可以是临时znode，由create -e 生成的节点，一旦创建这个znode的client与server断开连接，该znode将被自动删除。

client和server之间通过heartbeat来确认连接正常，这种状态称之为session，断开连接后session失效。

（3）临时znode不能有子znode。

（4）znode可以自动编号，由create -s 生成的节点，例如在 create -s /app/node 已存在时，将会生成 /app/node00\*\*\*001节点。

（5）znode可以被监控，该目录下某些信息的修改，例如节点数据、子节点变化等，可以主动通知监控注册的client。事实上，通过这个特性，可以完成许多重要应用，例如配置管理、信息同步、分布式锁等等。

### 3.2 ZooKeeper服务中的操作

在ZooKeeper中有9个基本操作，如下图所示：



更新ZooKeeper操作是有限制的。delete或setData必须明确要更新的Znode的版本号，我们可以调用exists找到。如果版本号不匹配，更新将会失败。

更新ZooKeeper操作是非阻塞式的。因此客户端如果失去了一个更新（由于另一个进程在同时更新这个Znode），他可以在不阻塞其他进程执行的情况下，选择重新尝试或进行其他操作。

尽管ZooKeeper可以被看做是一个文件系统，但是处于便利，摒弃了一些文件系统地操作原语。因为文件非常的小并且使整体读写的，所以不需要打开、关闭或是寻地的操作。

#### 3.2.1 watch触发器

读操作exists、getChildren和getData都被设置了watch，并且这些watch都由写操作来触发：create、delete和setData。ACL操作并不参与到watch中。当watch被触发时，watch事件被生成，他的类型由watch和触发他的操作共同决定。ZooKeeper所管理的watch可以分为两类：

1.数据watch（data watches）：getData和exists负责设置数据watch；

2.孩子watch（child watches）：getChildren负责设置孩子watch；

我们可以通过操作返回的数据来设置不同的watch：

1.getData和exists：返回关于节点的数据信息

2.getChildren：返回孩子列表

因此，一个成功的setData操作将触发Znode的数据watch。

一个成功的create操作将触发Znode的数据watch以及孩子watch。

一个成功的delete操作将触发Znode的数据watch以及孩子watch。

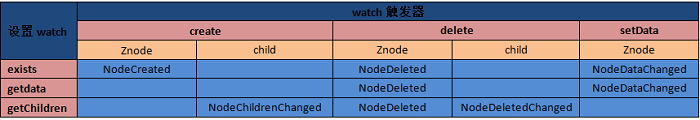
watch由客户端所连接的ZooKeeper服务器在本地维护，因此watch可以非常容易地设置、管理和分派。当客户端连接到一个新的服务器上时，任何的会话事件都将可能触发watch。另外，当从服务器断开连接的时候，watch将不会被接收。但是，当一个客户端重新建立连接的时候，任何先前注册过的watch都会被重新注册。

exists操作上的watch，在被监视的Znode创建、删除或数据更新时被触发。

getData操作上的watch，在被监视的Znode删除或数据更新时被触发。在被创建时不能被触发，因为只有Znode一定存在，getData操作才会成功。

getChildren操作上的watch，在被监视的Znode的子节点创建或删除，或是这个Znode自身被删除时被触发。可以通过查看watch事件类型来区分是Znode还是他的子节点被删除：NodeDelete表示Znode被删除，NodeDeletedChanged表示子节点被删除。

watch设置操作及相应的触发器如图下图所示：



watch事件包括了事件所涉及的Znode的路径，因此对于NodeCreated和NodeDeleted事件来说，根据路径就可以简单区分出是哪个Znode被创建或是被删除了。为了查询在NodeChildrenChanged事件后哪个子节点被改变了，需要再次调用getChildren来获得新的children列表。同样的，为了查询NodeDeletedChanged事件后产生的新数据，需要调用getData。在两种情况下，Znode可能在获取watch事件或执行读操作这两种状态下切换，在写应用程序时，必须记住这一点。

**（1）Zookeeper的watch实际上要处理两类事件：**

1. 连接状态事件(type=None, path=null)

这类事件不需要注册，也不需要我们连续触发，我们只要处理就行了。

2. 节点事件

节点的建立，删除，数据的修改。它是one time trigger，我们需要不停的注册触发，还可能发生事件丢失的情况。

上面2类事件都在Watch中处理，也就是重载的process(Event event)

**（2）节点事件的触发，通过函数exists,getData或getChildren来处理**

这类函数，有双重作用：

1. 注册触发事件

2. 函数本身的功能

函数的本身的功能又可以用异步的回调函数来实现,重载processResult()过程中处理函数本身的的功能。

函数还可以指定自己的watch,所以每个函数都有4个版本。根据自己的需要来选择不同的函数，不同的版本。

**3.3 ZooKeeper访问控制列表ACL**

ZooKeeper使用ACL来对Znode进行访问控制。ACL的实现和Unix文件访问许可非常相似：它使用许可位来对一个节点的不同操作进行允许或禁止的权限控制。但是，和标准的Unix许可不同的是，Zookeeper对于用户类别的区分，不止局限于所有者(owner)、组 (group)、所有人(world)三个级别。Zookeeper中，数据节点没有“所有者”的概念。访问者利用id标识自己的身份，并获得与之相应的 不同的访问权限。

**注意：**

传统的文件系统中，ACL分为两个维度，一个是属组，一个是权限，子目录/文件默认继承父目录的ACL。而在Zookeeper中一个ACL和一个ZooKeeper节点相对应。并且，父节点的ACL与子节点的ACL是相互独立的。也就是说，ACL不能被子节点所继承，父节点所拥有的权限与子节点所用的权限都没有任何关系。

Zookeeper支持可配置的认证机制。它利用一个三元组来定义客户端的访问权限：(scheme:expression, perms) 。其中：

1.scheme：定义了expression的含义。

如：（host:host1.corp.com，READ）,标识了一个名为host1.corp.com的主机,有该数据节点的读权限。

2.Perms：标识了操作权限。

如：（ip:19.22.0.0/16, READ）,表示IP地址以19.22开头的主机,有该数据节点的读权限。

Zookeeper的ACL也可以从三个维度来理解：一是，scheme; 二是，user; 三是，permission，通常表示为scheme:id:permissions，如下图所示。



1.world : id格式：anyone。

如：world:anyone代表任何人，zookeeper中对所有人有权限的结点就是属于world:anyone的。

2.auth : 它不需要id。

注：只要是通过authentication的user都有权限，zookeeper支持通过kerberos来进行认证, 也支持username/password形式的认证。

3.digest: id格式：username:BASE64(SHA1(password))。

它需要先通过username:password形式的authentication。

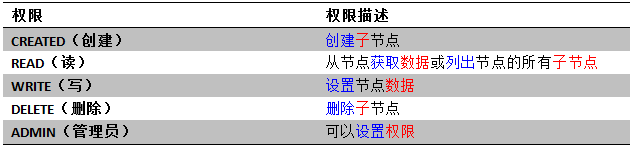
4.ip: id格式：客户机的IP地址。

设置的时候可以设置一个ip段。如：ip:192.168.1.0/16, 表示匹配前16个bit的IP段

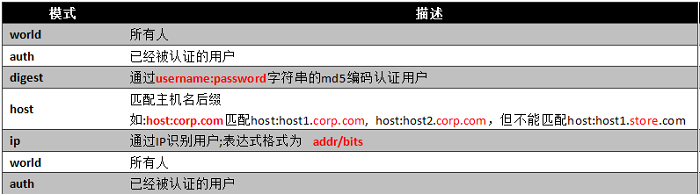
5.super: 超级用户模式。

在这种scheme情况下，对应的id拥有超级权限，可以做任何事情

ZooKeeper权限定义如下图所示：



ZooKeeper内置的ACL模式如下图所示：



当会话建立的时候，客户端将会进行自我验证。另外，ZooKeeper [**Java**](http://lib.csdn.net/base/java)API支持三种标准的用户权限，它们分别为：

1.ZOO\_PEN\_ACL\_UNSAFE：对于所有的ACL来说都是完全开放的，任何应用程序可以在节点上执行任何操作，比如创建、列出并删除子节点。

2.ZOO\_READ\_ACL\_UNSAFE：对于任意的应用程序来说，仅仅具有读权限。

3.ZOO\_CREATOR\_ALL\_ACL：授予节点创建者所有权限。需要注意的是，设置此权限之前，创建者必须已经通了服务器的认证。

下面演示一个通过digest（用户名密码的方式）为创建的节点设置ACL的例子，代码如下：

|  |
| --- |
| import org.apache.zookeeper.\*;  import org.apache.zookeeper.server.auth.DigestAuthenticationProvider;  import org.apache.zookeeper.data.\*;  import java.util.\*;  public class NewDigest {  public static void main(String[] args) throws Exception {//new一个acl  List acls = new ArrayList();  　　　　　//添加第一个id，采用用户名密码形式  Id id1 = new Id("digest",DigestAuthenticationProvider.generateDigest("admin:admin"));  ACL acl1 = new ACL(ZooDefs.Perms.ALL, id1);  acls.add(acl1);    　　　　 //添加第二个id，所有用户可读权限  Id id2 = new Id("world", "anyone");  ACL acl2 = new ACL(ZooDefs.Perms.READ, id2);  acls.add(acl2);  // zk用admin认证，创建/test ZNode。  　　　　　ZooKeeper zk = new ZooKeeper("host1:2181,host2:2181,host3:2181",2000, null);  　　　　　zk.addAuthInfo("digest", "admin:admin".getBytes());  　　　　　zk.create("/test", "data".getBytes(), acls, CreateMode.PERSISTENT);  }  } |

### 3.4 ZooKeeper的执行

ZooKeeper服务可以以两种模式运行。在单机模式下，只有一个ZooKeeper服务器，便于用来**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "_blank)**。但是他没有高可用性和恢复性的保障。在工业界，ZooKeeper以复合模式10运行在一组叫ensemble的集群上。ZooKeeper通过复制来获得高可用性，同时，只要ensemble中大部分机器运作，就可以提供服务。在2n+1个节点的ensemble中，可以承受n台机器故障。

ZooKeeper的思想非常简单：他所需要做的就是保证对Znode树的每一次修改都复制到ensemble中的大部分机器上去。如果机器中的小部分出故障了，那么至少有一台机器将会恢复到最新状态，其他的则保存这副本，直到最终达到最新状态。Zookeeper采用Zab协议，它分为两个阶段，并且可能被无限的重复。

#### （1）阶段1：领导者选举

在ensemble中的机器要参与一个选择特殊成员的进程，这个成员叫领导者，其他机器脚跟随者。在大部分的跟随者与他们的领导者同步了状态以后，这个阶段才算完成。

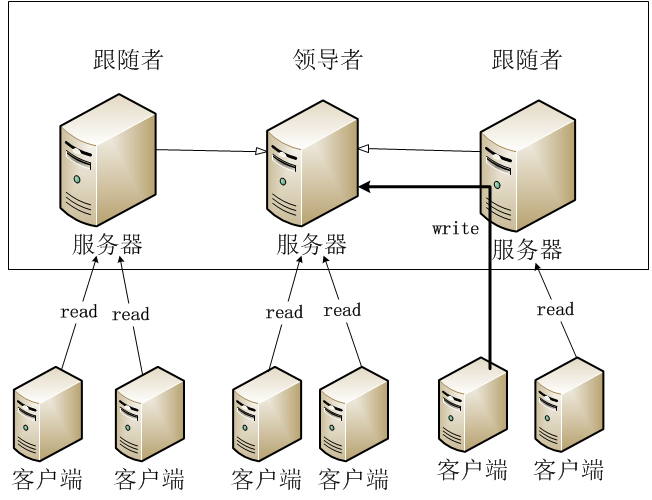
#### （2）阶段2：原子广播

所有的写操作请求被传送给领导者，并通过广播将更新信息告诉跟随者。当大部分跟随者执行了修改之后，领导者就提交更新操作，客户端将得到更新成功的回应。未获得一致性的协议被设计为原子的，因此无论修改失败与否，他都分两阶段提交。

如果领导者出故障了，城下的机器将会再次进行领导者选举，并在新领导被选出前继续执行任务。如果在不久后老的领导者恢复了，那么它将以跟随者的身份继续运行。领导者选举非常快，由发布的结果所知，大约是200毫秒，因此在选举是性能不会明显减慢。  
所有在ensemble中的机器在更新它们内存中的Znode树之前会先将更新信息写入磁盘。读操作请求可由任何机器服务，同时，由于他们只涉及内存查找，因此非常快。

### 3.5 ZooKeeper一致性

在ensemble中的领导者和跟随着非常灵活，跟随者通过更新号来滞后领导者11，结果导致了只要大部分而不是所有的ensemble中的元素确认更新，就能被提交了。对于ZooKeeper来说，一个较好的智能模式是将客户端连接到跟着领导者的ZooKeeper服务器上。客户端可能被连接到领导者上，但他不能控制它，而且在如下情况时，甚至可能不知道。参见下图：



每一个Znode树的更新都会给定一个唯一的全局标识，叫zxid（表示ZooKeeper事务“ID”）。更新是被排序的，因此如果zxid的z1<z2，那么z1就比z2先执行。对于ZooKeeper来说，这是分布式系统中排序的唯一标准。

ZooKeeper是一种高性能、可扩展的服务。ZooKeeper的读写速度非常快，并且读的速度要比写快。另外，在进行读操作的时候，ZooKeeper依然能够为旧的数据提供服务。这些都是由ZooKeeper所提供的一致性保证的，它具有如下特点：

#### （1）顺序一致性

任何一个客户端的更新都按他们发送的顺序排序，也就意味着如果一个客户端将Znode z的值更新为值a，那么在之后的操作中，他会将z更新为b，在客户端发现z带有值b之后，就不会再看见带有值a的z。

#### （2）原子性

更新不成功就失败，这意味着如果更新失败了，没有客户端会知道。☆☆

#### （3）单系统映像☆

无论客户端连接的是哪台服务器，他与系统看见的视图一样。这就意味着，如果一个客户端在相同的会话时连接了一台新的服务器，他将不会再看见比在之前服务器上看见的更老的系统状态，当服务器系统出故障，同时客户端尝试连接ensemble中的其他机器时，故障服务器的后面那台机器将不会接受连接，直到它连接到故障服务器。

#### （4）容错性☆☆☆

一旦更新成功后，那么在客户端再次更新他之前，他就固定了，将不再被修改，这就会保证产生下面两种结果：

如果客户端成功的获得了正确的返回代码，那么说明更新已经成功。如果不能够获得返回代码（由于通信错误、超时等原因），那么客户端将不知道更新是否生效。

当故障恢复的时候，任何客户端能够看到的执行成功的更新操作将不会回滚。

#### （5）实时性☆☆

在任何客户端的系统视图上的的时间间隔是有限的，因此他在超过几十秒的时间内部会过期。这就意味着，服务器不会让客户端看一些过时的数据，而是关闭，强制客户端转到一个更新的服务器上。

**解释一下：**

由于性能原因，读操作由ZooKeeper服务器的内存提供，而且不参与写操作的全局排序。这一特性可能会导致来自使用ZooKeeper外部机制交流的客户端与ZooKeeper状态的不一致。举例来说，客户端A将Znode z的值a更新为a'，A让B来读z，B读到z的值是a而不是a’。这与ZooKeeper的保证机制是相容的（不允许的情况较作“同步一致的交叉客户端视 图”）。为了避免这种情况的发生，B在读取z的值之前，应该先调用z上的sync。Sync操作强制B连接上的ZooKeeper服务器与leader保 持一致这样，当B读到z的值时，他将成为A设置的值（或是之后的值）

**容易混淆的是：**

sync操作只能被异步调用12。这样操作的原因是你不需要等待他的返回，因为ZooKeeper保证了任何接下去的操作将会发生在sync在服务器上执行以后，即使操作是在sync完成前被调用的。

这些已执行的保证后，ZooKeeper更高级功能的设计与实现将会变得非常容易，例如：leader选举、队列，以及可撤销锁等机制的实现。

### 3.6 ZooKeeper会话

ZooKeeper客户端与ensemble中的服务器列表配置一致，在启动时，他尝试与表中的一个服务器相连接。如果连接失败了，他就尝试表中的其他服务器，以此类推，知道他最终连接到其中一个，或者ZooKeeper的所有服务器都无法获得时，连接失败。

一旦与ZooKeeper服务器连接成功，服务器会创建与客户端的一个新的对话。每个回话都有超时时段，这是应用程序在创建它时设定的。如果服务器没有在超时时段内得到请求，他可能会中断这个会话。一旦会话被中断了，他可能不再被打开，而且任何与会话相连接的临时节点都将丢失。

无论什么时候会话持续空闲长达一定时间，都会由客户端发送ping请求保持活跃（犹如心跳）。时间段要足够小以监测服务器故障（由读操作超时反应），并且能再回话超市时间段内重新连接到另一个服务器。

在ZooKeeper中有几个time参数。tick time是ZooKeeper中的基本时间长度，为ensemble里的服务器所使用，用来定义对于交互运行的调度。其他设置以tick time的名义定义，或者至少由它来约束。

创建更复杂的临时性状态的应用程序应该支持更长的会话超时，因为重新构建的代价会更昂贵。在一些情况下，我们可以让应用程序在一定会话时间内能够重启，并且避免会话过期。（这可能更适合执行维护或是升级）每个会话都由服务器给定一个唯一的身份和密码，而且如果是在建立连接时被传递给ZooKeeper的话，只要没有过期它能够恢复会话。

这些特性可以视为一种可以避免会话过期的优化，但它并不能代替用来处理会话过期。会话过期可能出现在机器突然故障时，或是由于任何原因导致的应用程序安全关闭了，但在会话中断前没有重启。

### 3.7 ZooKeeper实例状态

Zookeeper对象的转变是通过其生命周期中的不同状态来实现。可以使用getState()方法在任何时候去查询他的状态：

|  |
| --- |
| public states getState() |

Zookeeper状态事务，如图3.5所示

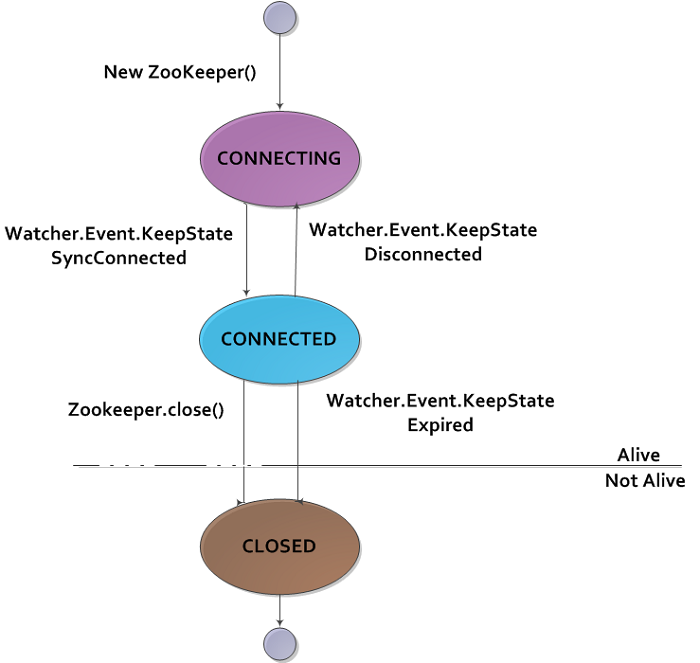


图 3.5 Zookeeper状态事务

　　getState()方法的返回类型是states，states是枚举类型代表Zookeeper对象可能所处的不同状态，一个Zookeeper实例可能一次只处于一个状态。一个新建的Zookeeper实例正在于Zookeeper服务器建立连接时，是处于CONNECTING状态的。一旦连接建立好以后，他就变成了Connected状态。

使用Zookeeper的客户端可以通过注册Watcher的方法来获取状态转变的消息。一旦进入了CONNNECTED状态，Watcher将获得一个KeepState值为SyncConnected的WatchedEvent。

**注意Zookeeper的watcher有两个职责：**

<1>了解Zookeeper的状态改变。传递给ZooKeeper对象构造函数的(默认)watcher，被用来监测状态的改变。

<2>了解Zonde的改变。监测Zonde的改变既可以使用专门的实例设置到读操作上，也可以使用读操作的默认watcher。

Zookeeper实例可能失去或重新连接Zookeeper服务，在CONNECTED和CONNECTING状态中切换。如果连接断开，watcher得到一个Disconnected事件。学要注意的是，这些状态的迁移是由Zookeeper实例自己发起的，如果连接断开他将自动尝试自动连接。

如果任何一个close()方法被调用，或是会话由Expired类型的KeepState提示过期时，ZooKeeper可能会转变成第三种状态CLOSED。一旦处于CLOSED状态，Zookeeper对象将不再是活动的了(可以使用states的isActive()方法进行测试)，而且不能被重用。客户端必须建立一个新的Zookeeper实例才能重新连接到Zookeeper服务。